A PLAIN-TEXT COMPRESSION TECHNIQUE WITH FAST LOOKUP ABILITY

A Thesis

by

HARSH KUMAR

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Sunil Khatri
Committee Members,	Paul Gratz
	Aakash Tyagi
	Jun Kameoka
Head of Department,	Miroslav M. Begovic

December 2016

Major Subject: Computer Engineering

ABSTRACT

Data compression has always been an essential aspect of computing. In recent times, with the increasing popularity of remote and cloud-based computation, compression is becoming more important. Reducing the size of a data object in this context would not only reduce the transfer time, but also the amount of data transferred. The key figures of merit of a data compression scheme are its compression ratio and its compression, decompression and lookup speeds. Traditional compression techniques achieve high compression ratios, but require decompression before a lookup can be performed. This increases the lookup time. In this thesis, we propose a compression technique for plain-text data objects, that uses variable length encoding to compress data. The dictionary of possible words is sorted based on the statistical frequency of the use of words, which are encoded using the variable length code-words. Words that are not in the dictionary are handled as well. The driving motivation of our technique is to perform significantly faster lookups without the need to decompress the compressed data object. Our approach also facilitates string operations (such as concatenation, insertion and deletion and search-and-replacement) on compressed text without the need of decompression. We implement our technique in C++, and compare our approach with industry standard tools like gzip and bzip2 in terms of compression ratio, lookup speed, search-and-replace time and peak memory uses. Our compression scheme is about 81× faster as compared to gzip and about 165× times faster as compared to bzip2, when the data is searched, and restored into a compressed format. In conclusion, Our approach facilitates string operations like concatenation, insertion, deletion and search-and-replace on the compressed file itself without the need for decompression.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Dr. Sunil Khatri for the continuous support. His active advice and continuous suggestions have prepared me to become a better researcher. I am so thankful to him for his guidance.

Besides my advisor, I would like to thank the rest of my advisory committee: Dr. Aakash Tyagi, Dr. Paul Gratz, and Dr. Jun Kameoka, for their support and enthusiasm.

I would like to express my heartfelt thanks to my family for their love and support. I would like to thank my group mates in SPK group for their continuous support and motivation. I would specially like to thank Monther for his valuable help whenever needed. I would like to acknowledge Ahmad for his help in coming up with an approach to eliminate false negatives in search.

Finally, I would like to thank my roommates and friends at College Station for their moral support.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION

Data compression is a technique by which a data object F is encoded into another data object F_C, such that F_C utilizes fewer bits in its representation. The ratio $\frac{F}{F_C}$ is referred to as the *compression ratio* of the data compression scheme, and is an important metric used for comparing different data compression schemes.

Data compression schemes are classified as either a) *lossless* or b) *lossy*. In lossless schemes, no information is lost during the decompression phase, and F is reconstructed exactly after decompression. Examples of lossless compression are text or data compression. In lossy schemes, some information is lost during decompression, and F is not reconstructed exactly after decompression. Video and audio are representative examples where lossy compression is used.

Consider lossless compression schemes for plain-text data objects F. First, we would compress F to produce F_C as shown in Figure 1.1 (a). In such schemes, it is often the case that a user would like to test if a data object S is present in F. Traditionally, the user would decompress F_C to reconstruct F, and then test if S is contained in F. This is illustrated in Figure 1.1 (b).

The goal of this thesis is to design and test a lossless compression scheme for plain-text data objects F, such that the test of whether S is contained in F is done without requiring F to be reconstructed from F_C (in other words, decompression of F_C to reconstruct F is not required). Our scheme first converts S into an encoded (compressed) data object S_C, and then checks if S_C is contained in F_C. Since S is generally much smaller than F_C in plain-text search operations, this saves a significant amount of time, since the decompression of F_C to reconstruct F is avoided. This is illustrated in Figure 1.2 (b). In the sequel, we will equivalently call the search

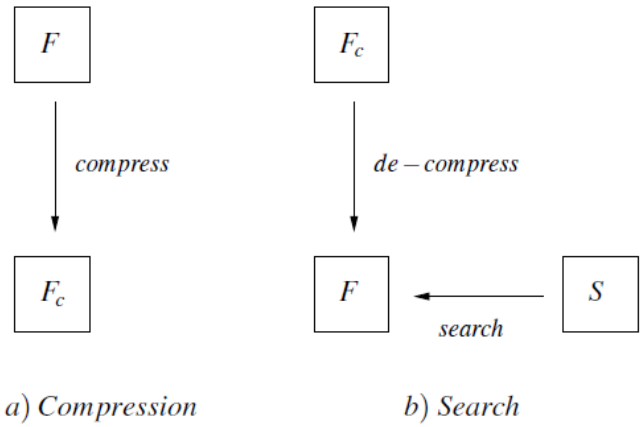

Figure 1.1: Traditional Compression and Search

operation (i.e. the test of whether S is contained in F) as the *lookup operation*.

The search (lookup) operation occurs frequently in practice, and is rendered inefficient in the traditional lossless compression scenario, especially when F is large and S is small, which occurs frequently. There are several scenarios where such a search occurs. For example, in online search, the index of pages that contain a specific keyword is usually stored in a compressed fashion, since the number of such indices is extremely large. When a search query is initiated for a particular string, the indices of the keywords in the string are retrieved and de-compressed, after which an intersection is performed to obtain the pages containing all the keywords in the string. This intersected list is displayed to the user, in rank order. If the intersection could be performed on the compressed indices, search could be sped up. Another example is from the data science field and the emerging field of big data. When a database is extremely large, it is inevitably stored in a compressed manner. Searching a string in the compressed database is inefficient, since the database needs to be

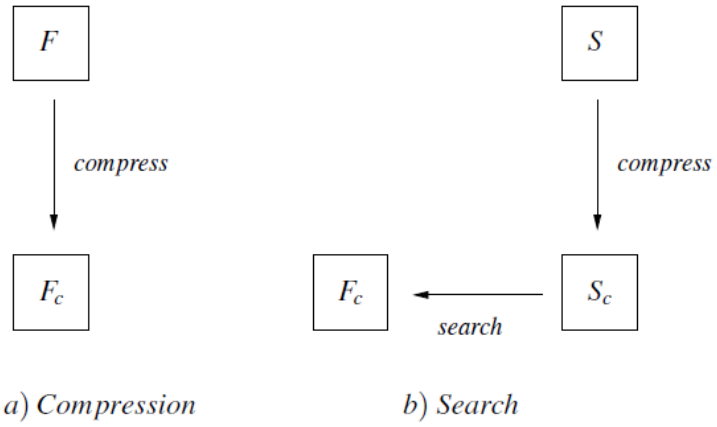

Figure 1.2: Compression and Search in Our Scheme

de-compressed before the search query can be performed. Another example, when large amounts of data are stored in a distributed manner on the cloud, they are stored in a compressed manner in order to reduce the time and bandwidth used to transfer them. Again, if a search is to be done on such compressed data, the need to decompress causes the search to be slowed down. In many cases, the object being searched in the above scenarios is a plain-text (ASCII) data object. Finally in web-based systems, if multiple users want to search a string S, each of the users will have to decompress the file and perform the search on the decompressed data. The proposed compression scheme would be very well suited in such scenarios, and will allow the users to directly perform the search on the compressed data. This is an emerging class of usage scenarios in widely used applications.

Given the occurrence frequency of the words in any language, we first sort the words in decreasing order of occurrence frequency. Punctuation characters are included in this dictionary as well. We do not include phrases in this list. Then, unlike existing compression techniques which utilize code-words of fixed length, we encode

these dictionary words with code-words which have variable lengths. Non-dictionary words are handled in our scheme as well. The varying length code-words are constructed in a manner that eliminates false positives and false negatives during the search or lookup operation. A false positive is said to have occurred when a word doesn't exist in a text file yet the search result returns "true". A false negative is said to have occurred when a word exists in a text file yet the search result returns "false". We verify that our compressed file F_C can be de-compressed to yield F in a lossless manner, and compare our scheme against commonly deployed compression techniques. Our compression technique achieves a search time that is linear in the size of the compressed file F_C. Our compression scheme is about 81× faster as compared to gzip and about 165× times faster as compared to bzip2, when the data is searched, and restored into a compressed format.

The key contributions of this thesis are:

- We present an end-to-end data compression methodology to perform lossless compression, de-compression and search which is targeted for plain-text data objects.

- Our compression methodology allows search operations to be directly performed on compressed data objects, significantly dropping search times.

- Our methodology uses code-words that are constructed in a way that eliminates aliasing and false positives.

- Our methodology also facilitates string operations to be performed on compressed text (concatenation, insert, delete, search-and-replace).

- We initially sort all the words (including punctuation symbols) of a given language in decreasing order of occurrence. While encoding any word W of a data

file, we use smaller code-words if the rank of W in the sorted list is high, and longer code-words if the rank of W is low.

- We test our scheme on a set of readily available text files, and over a set of examples, and compare our scheme with the popularly used compression tools like gzip and bzip2. Our scheme achieves a compression ratio of 1.94 compared to 2.72 and 3.89 for gzip and bzip2 respectively.

The rest of this thesis is organized as follows. Section 2 discusses previous work in this area. In Section 3, we present our approach, while Section 4 presents our experiments and results. In Section 5, we provide our conclusions. Finally we discuss future work in Section 6.

2. PREVIOUS WORK

2.1 Lossless Compression Techniques

There has been a great deal of work on lossless data compression. These compression techniques achieve healthy compression ratio but fails to perform a search over compressed file. Some of the common classes of these compression methods are the following.

(a) Huffman coding [8] and its derivatives use fewer bits to encode high frequency characters. In this technique the text file is scanned first and the frequency of occurrence of each character in the file is calculated. Finally, a shorter bit sequence is assigned for high frequency characters. The assigned codes are designed to be prefix free. Prefix codes are codes that don't overlap with the prefix of any other code.

(b) Arithmetic coding and its derivatives [17, 16] are a form of entropy encoding. This algorithm takes a stream of input symbols and encodes an entire string into a single fraction number n, where $(0.0 \leq n \leq 1.0)$. For example suppose we have to compress string 'B' from the alphabet A,B,C. Now suppose all characters have the same probability of occurrence i.e P[A]=P[B]=P[C]= 0.33. The algorithm divides the probability space from 0 to 1 into 3 equal parts. Now 'A' can be encoded as any fractional number between 0 to 0.33. 'B' can be encoded as any fraction between .3333 to .6667 and 'C' can be compressed using any fraction from .6667 to 1. The interested reader can go through the reference [17, 16] for further detail.

(c) Dictionary based methods, exemplified by Lempel-Ziv-Welch [22, 23, 20] scan

the input file and create a dictionary on-the-fly assigning shorter code for the repeated occurrences of text data. It finally replaces repeated occurrences of data with codes in the dictionary.

(d) Run-length encoding [18] is used for repeated data values. It stores a single data value and a repetition count, rather than the original run of data. For example run-length encoding will encode the string $xxxxxxxxxyyyy$ as $10x4y$.

(e) The Burrows-Wheeler transform (BWT) [4, 13] is used in bzip2. The BWT algorithm permutes the order of the characters followed by sorting the different permutations as per the first character in the permutation. It finally compress the word using the last characters of the permutation.

2.2 Lookup Techniques on Compressed Data

There have been several efforts to compress plain-text data using dictionary words, to allow for quick search. These efforts fall under a class of techniques called *compressed pattern matching*. Amir and Benson [1] introduced the problem of compressed pattern matching. They search a two dimensional pattern on the compressed digital image file that is compressed using two-dimensional run-length compression technique [18] . Since then, there has been lots of work in this field.

The authors of [2] use 19-bit fixed code-words to encode all available English words to pre-compress a text file. They further compress the pre-compressed file using the *deflate* compression algorithm, which uses Huffman coding [8] as well as a LZ-77 [22] algorithm. The pre-compression ratio achieved in the compression step is slightly over $2\times$. However this work is not focused on searching texts. Since they are using 19-bit codes, patterns have to be aligned at 19 bit boundaries, which may consequently produce false positives or false negatives.

In [7], the authors use a block dictionary-tree based scheme to compress data, which is derived from the LZW [22, 23, 20] algorithm (which is used in GIF images), with some modification to do compressed pattern matching. The search is conducted one character at a time , and is slow.

In [9] the author uses a dictionary size of 40K words to compress text files. They use 2 byte fixed encoding. However, this approach produces false negatives. For example assume we have a file F which has the string "We are going to school". Now if we search the pattern S "going to", the approach in [9] may produce a false negative due to the fact that phrases are encoded in their compression scheme.

The author in [10] presents an algorithm which minimizes false positives and false negatives. The algorithm is focused on the synchronization of a pattern in a file that is compressed using static Huffman coding technique. However our algorithm completely eliminates false positives or false negatives. In [3] the author use a dynamic dictionary to pre-compress the file on the fly. The resulting file is suitable for searching strings. However, the use of a dynamic dictionary forces the dictionary to be stored along with the pre-compressed result. This degrades the compression ratio and results in additional memory overhead. Our method does not suffer from this drawback since it doesn't require storing a dictionary with the compressed data.

The authors in [14] split a big file into small files (l_1 to l_n). They preprocess the files to find appropriate compression algorithm i.e. lossy compression for image, lossless compression for texts. They finally decompress a file (l_i) from the pool of files (l_1 to l_n), rather than decompressing the big file to search a string, where i is determined using indexing.

3. OUR APPROACH

In our approach, we first construct a list D of words of the (English) language, sorted in decreasing order of their frequency of occurrence. In our implementation, we construct a file containing D such that there is only one word per line. Punctuation symbols are included in this sorted list. A word W that occurs in the n^{th} line of D is said to have a *rank* of n. In other words, *rank(W)* $= n$. If two words W_1 and W_2 have the ranks n_1 and n_2 respectively, and $n_1 < n_2$, then n_1 is said to have a higher rank than n_2. Note that our list D does not include n-grams or phrases. Only words and punctuation symbols are supported. Also, every word in D is assumed to be lower-case. Our list D also contains digits from 0 to 9, alphabets from a to z and A to Z. We will explain its reason later in the discussion.

Now consider a data object (text file) F, which we intend to compress into a compressed data object F_C. The compressed file F_C is constructed using variable length code-words, which are described later in this section. Compression proceeds in the following manner:

- For every word W in F, we find *rank(W)* from D, and append the code-word C corresponding to *rank(W)* to F_C. Since we use variable length code-words, if *rank(W)* is high, a shorter code-word will be utilized.

- While compressing F, every word W is implicitly assumed to have a space character preceding it. If a word W does not have a space character preceding it (for example ",hello"), then a special code for a negative space is inserted in F_C, before the code for W. The de-compression algorithm accounts for this situation as well.

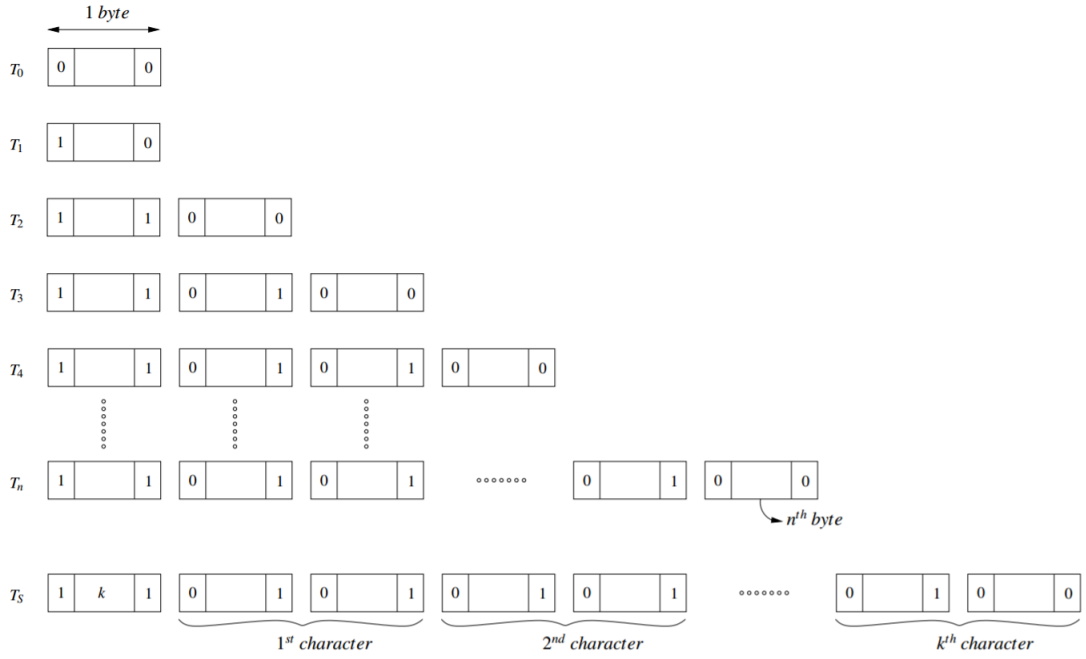

Figure 3.1: Code-Words Used in Our Scheme

- The file F consists of several words, some of which may not appear in the list D. These are handled with special code-words, as will be described later.

- An upper-case word W in F is handled with a special code-word that precedes the code-word C corresponding to the word W. Every word in D is lower-case. A word W which has mixed capitalization (for example "BuBBle") is treated as if it did not exist in D (see previous item).

In Figure 3.1, we illustrate the different kinds of code-words supported by our scheme. The code-words are referred to as T_0, T_1, T_2, T_3, $\cdots$, T_n, T_S. In our implementation, we use $n = 4$.

The meaning of each of these code-words is briefly described below, following which we discuss their construction. Each code-word is comprised of an integer number of bytes.

- T_0 (1 byte): This code-word indicates one of three conditions – a) whether the next word is capitalized, b) whether a negative space should be inserted before the next word, or c) whether a newline should be inserted.

- T_1 (1 byte): This code-word represents the codes for the highest ranked words in D. This code word handles words of rank $1 \leq r \leq R_1$, where R_i is a function of the number of bits in each code-word.

- T_2 (2 bytes): This code-word represents the codes for words of rank $R_1 + 1 \leq r \leq R_2$.

- T_3 (3 bytes): This code-word represents the codes for words of rank $R_2 + 1 \leq r \leq R_3$.

- T_4 (4 bytes): This code-word represents the codes for words of rank $R_3 + 1 \leq r \leq R_4$.

- $T_5, T_6, \cdots, T_{n-1}$ (5, 6, $\cdots$, n-1 bytes respectively): These code-words represent the codes for words of rank $R_4 + 1 \leq r \leq R_5$, $R_5 + 1 \leq r \leq R_6$, $\cdots$, $R_{n-2} + 1 \leq r \leq R_{n-1}$ respectively.

- T_n (n bytes): This code-word represents the codes for words of rank $R_{n-1} + 1 \leq r \leq R_n$.

- T_S ($2k$ + 1 bytes): This code-word represents the codes for *special* words (i.e. words that do not appear in D). For such a word W with k characters, T_S will have a length of $2k$ + 1 bytes. The first byte contains 6 data bits which encode the length of the word being represented. Then, the ASCII code for the first character of the word is stored in the 12 data bits of bytes 2 and 3 (4

bits are unused). The second character of the word is stored in bytes 4 and 5. In general, the i^{th} character of the word is stored in bytes $2 \cdot i$ and $2 \cdot i + 1$.

Consider codes T_1 through T_n in Figure 3.1, we note that the first bit of the first byte of any code is a '1' (start bit), and the last bit of the last byte of any code is a '0' (end bit). For example, for T_0, the first bit is '1', and the last bit is '0', indicating it is a one-byte code. For T_i $(i > 1)$, the first bit of any byte $j \geq 2$ is '0', indicating that this byte does not mark the start of a new code. Again, for T_i $(i > 1)$, the last bit of any byte $1 \leq j < i$ is a '1', indicating that this byte does not mark the end of a new code. Only the last bit of the byte i is a '0', indicating that the code ends with byte i.

Theorem 1. *Any word W, when searched in F_C, cannot generate a false positive.*

Proof. We start the search process by encoding W into its corresponding code-word C^*.

Now let's consider the following cases.

Case 1: Assume that W is encoded as a code-word C^* of type T_1 before being searched in F_C.When searching for C^*, clearly it cannot match any other code of type T_1 in F_C other than itself. It also cannot match any code C of type T_2, since the first byte of C has a '1' in the last bit position, unlike C^*, and the second byte of C has a '0' in the first bit position, unlike C^*. Hence any code of type T_1 cannot match a code of type T_2 in F_C. By a similar argument, a code of type T_2 cannot match a code of type T_1 in F_C. Also, any codes of type T_1 and T_j $(2 \leq j \leq n)$ cannot match.

Case 2: Assume that W is encoded as a code-word C^* of type T_2 before being searched in F_C. Clearly it cannot match any other code of type T_2 in F_C other than itself. It cannot match any code C of type T_3, since C^* and C both have the first

(last) bytes beginning and ending '1' (beginning and ending in '0'), however C has an intervening byte which begins in '0' and ending in '1', making the match impossible.

Case 3: Using the argument from the previous case, we can say that in general for a word W, which is encoded as a code-word C^* of type T_i before being searched in F_C, cannot match any code C of type T_j ($j \neq i$, $j, i \geq 2$). □

Theorem 2. *Any word W, when searched in F_C, cannot generate a false negative.*

Proof. We start the search process by encoding W into its corresponding code-word C^*. Since W exists in F, then the code C^* for W would be present in F_C. When W is searched, the code C^* will be included in S_C, ensuring that a match will occur. □

So far, we discussed codes T_1 through T_n, and shown that they cannot generate false positives or false negatives during a search. Now let us consider T_0. Since the first and last bits of T_0 are '0', it cannot match *any* byte of codes T_1 through T_n.

Finally, let us consider a code-word of type T_S. Note that if such a code-word was used for a special word of 2 characters or more, it would be 5 or more bytes long. As a result, for reasons discussed above, it cannot generate false positives or false negatives with any of the previously discussed code-words, since $n = 4$. However, if a code-word of type T_S represents a special word which is one character long, it uses 3 bytes. In this case, it could overlap a code-word of type T_3. To avoid this situation, we ensure that any code-word of type T_S will never represent a special word which is one character long, by adding all the singleton letters of the alphabet in D, which will ensure that they are encoded with a code of type T_1, T_2, T_3, or T_4.

We note that codes of type T_1 can represent 2^6 words (with ranks $1 \leq r \leq 2^6$). Similarly, codes of type T_2 can represent 2^{12} words (with ranks $2^6+1 \leq r \leq 2^6+2^{12}$). Codes of type T_3 can represent 2^{18} words (with ranks $2^6+2^{12}+1 \leq r \leq 2^6+2^{12}+2^{18}$). Finally, codes of type T_4 can represent 2^{24} words (with ranks $2^6 + 2^{12} + 2^{18} + 1 \leq$

$r \leq 2^6 + 2^{12} + 2^{18} + 2^{24}$). The total number of dictionary words that our scheme can support with $n = 4$ is $2^6 + 2^{12} + 2^{18} + 2^{24}$, which is 17,043,520. Recall that we can use T_S to cover any word which is outside of our list of dictionary words D.

Theorem 3. *The concatenation of two compressed files F_{C1} and F_{C2} is equivalent to concatenating the files $F1$ and $F2$ then applying our compression algorithm on the resulting file, assuming that $F1$ and $F2$ are terminated with punctuation or newline.*

Proof. Assuming we have a file $F1$ and we compress it into F_{C1}, similarly $F2$ would produce F_{C2}. If we concatenate $F1$ and $F2$ into $F3$ given that no new words are generated from the concatenation (since $F1$ and $F2$ are terminated using a punctuation or newline), then the compressed version of $F3$ (F_{C3}) will have the codewords of F_{C1} followed by the code-words of F_{C2}, since we are using the same dictionary when compressing any file. □

4. EXPERIMENTS AND RESULTS

4.1 Experimental Results

In this section, we first discuss the construction of the word list D, and then we discuss the results from our experiments.

The compression, decompression and search code is implemented in C++. The search code invokes a python script to perform hex pattern search on a compressed file. The python script is called "SearchBin" [19].

Before running our benchmarking experiments, we verified that our code is able to compress a plain-text file F into a file F_C, which when de-compressed results in a file F_D, such that F and F_D are identical. All simulations are performed on a 2.67GHz Linux server with 8MB of L2/L3 cache and a total of 8 cores. The machine has 9GB DDR2-1066 RAM. Although the machine supports multithreading, our the code does not.

4.1.1 Constructing the Sorted Wordlist D

We obtained a list of words of the English language, along with their frequency of occurrence from [21]. Similarly, we obtained a list of punctuation characters, along with their occurrence frequency from [15]. We merged these two lists, and sorted them in decreasing order of occurrence frequency to create the sorted wordlist D which is used for all our subsequent experiments. Note that all ASCII characters are added to this sorted wordlist, in order to eliminate the possibility that the compression algorithm would choose code-words of type T_S for single-character special words. These characters are given ranks such that they would generate codes of type T_3.

4.1.2 Results

Our scheme is benchmarked using several plain-text files, which were generated by combining files from the Gutenburg corpus [6]. The files in this corpus were small, so we concatenated several files to generate our benchmark examples. We compare our results against two popular compression schemes, gzip and bzip2. For each data point, we run the experiment 10 times and get the average of all runs. When we repeat our experiment 10 times, we make sure that neither our algorithm nor gzip and bzip2 benefit from caching.

The first experiment is run to find the compression ratio of all 3 schemes. The result of this experiment is shown in Table 4.1. The compression ratio is defined as $CR = \frac{size(original\ \ file)}{size(compressed\ \ file)}$. On average our approach achieves a compression ratio of about 1.94 while gzip achieves a compression ratio of about 2.72 and bzip2 achieves a compression ratio of about 3.89.

File Name	Original Size (Bytes)	Compression Ratio (CR)		
		gzip	bzip2	Our Approach
f1.txt	52428800	2.73	3.96	2.11
f2.txt	104857600	2.7	3.92	1.86
f3.txt	143774120	2.96	3.8	1.7
f4.txt	157286400	2.72	3.96	1.87
f5.txt	209715200	2.67	3.67	1.91
f6.txt	364366412	2.69	3.89	2.04
f7.txt	419430400	2.7	3.93	1.99
f8.txt	524288000	2.69	3.96	1.92
f9.txt	665008138	2.68	3.88	1.98
f10.txt	806449643	2.69	3.89	1.97
Average	344760471	2.72	3.89	1.94

Table 4.1: Compression Ration for all Schemes

In Figure 4.1, we report the histogram of code-words used to compress the same

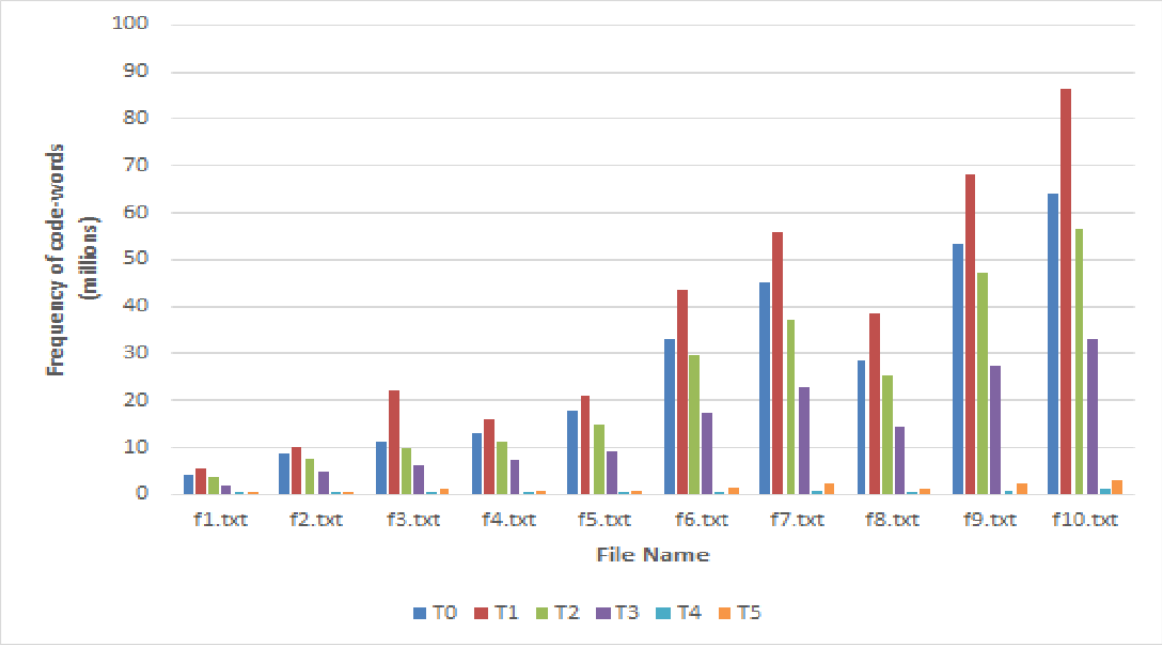

Figure 4.1: Histogram of Code-Words

files shown in Table 4.1. The x-axis in Figure 4.1 shows the files that we compressed using our approach, and the y-axis shows the frequency of each code-word used in the compression process. The figure shows that for each compressed file, T_0 and T_1 are the most occurring code-words, T_2 has the next highest frequency followed by T_3, T_S and T_4 respectively. Note that T_0 and T_1 are both 1-byte each (which is the smallest code-word size used in our approach) resulting in the highest compression ratio since they have the highest frequency.

Similarly T_2 is a 2-byte code-word and has the next highest frequency followed by T_3 and T_4 which are 3-byte and 4-byte code-words respectively. We also note that T_S which is a variable code-word (with a minimum of 5 bytes), has a very low occurrence rate.

Table 4.2 reports the run-time to perform a search on a compressed file F_C. In the case of gzip and bzip2 tools, we report the run-time needed to decompress F_C to

obtain F in addition to the search runtime (which is performed on F). However, for our approach recall that we can search a pattern in F_C directly, so we only report the search run-time. Table 4.2 shows that on average our approach is about 17.45× faster than gzip and is about 48.77× faster than bzip2.

File Name	Absolute (s)	Relative to our approach	
	Our Approach	gzip	bzip2
f1.txt	0.10	9.77	26.40
f2.txt	0.20	10.12	28.38
f3.txt	0.16	16.33	46.27
f4.txt	0.27	11.05	30.63
f5.txt	0.21	19.87	56.22
f6.txt	0.31	22.47	62.016
f7.txt	0.39	20.51	57.21
f8.txt	0.49	20.6	56.36
f9.txt	0.56	22.59	64.41
f10.txt	0.72	21.22	59.89
Average	0.34	17.45	48.77

Table 4.2: Search Time (without Re-compression)

Table 4.3 also reports the search run-time. However, unlike the results shown in Table 4.2, in Table 4.3 we also account for the re-compression run-time in the case of gzip and bzip2 tools. This consideration is necessary for systems that have limited space and cannot hold both the compressed and uncompressed versions of a file. The search time for our approach in Table 4.3 remains the same as that in Table 4.2 since our approach can perform the search on a compressed file directly. On average, our approach is about 81× and 165× faster than gzip and bzip2 respectively.

In Table 4.4 we report the relative speedup in the search time as a function of file size. For gzip and bzip2, we first de-compress a compressed file and perform a look-up. However for our approach we perform the search directly. We observe that for

File Name	Absolute (s)	Relative to our approach	
	Our Approach	gzip	bzip2
f1.txt	0.10	44.57	90.24
f2.txt	0.20	48.44	97.84
f3.txt	0.16	72.01	153.60
f4.txt	0.27	51.60	104.97
f5.txt	0.21	93.47	186.08
f6.txt	0.31	104.45	208.72
f7.txt	0.39	96.94	196.027
f8.txt	0.494	94.71	192.16
f9.txt	0.56	106.89	216.46
f10.txt	0.72	99.33	201.26
Average	0.34	81.24	164.73

Table 4.3: Search Time (with Re-compression)

smaller file sizes (upto 200MB), the speed up rate increases as the file size increases, and for file sizes greater than 200MB, the search speed rate stays approximately constant.

Table 4.5 reports the peak memory utilization for de-compression, search and re-compression using gzip and bzip2 tools. For our approach, the peak memory reported is that for performing the search on the compressed file directly. Table 4.5 shows that our approach uses about 10-27× more memory than bzip2 and gzip. The high memory utilization is due to loading the entire dictionary D into memory to perform code-word lookup.

Table 4.6 and Table 4.7 report the compression and the de-compression times respectively using our approach, gzip and bzip2. On average, the compression times for gzip and bzip2 tools are about 0.30× and 0.54× the compression time of our approach. Also the decompression times of gzip and bzip2 are about 0.091× and 0.325× the decompression time of our approach.

In Table 4.8, we report search time of our approach with Linux commands zgrep

File Size (MB)	Relative to our approach	
	gzip	bzip2
50	9.77	26.40
100	10.12	28.38
137	16.33	46.27
150	11.05	30.63
200	19.87	56.22
347	22.47	62.02
400	20.51	57.21
500	20.60	56.36
634	22.59	64.41
769	21.22	59.89

Table 4.4: Search Time Speedup with File Size

[12] and bzgrep [11]. zgrep and bzgrep are the Linux commands to search text data on compressed files for gzip and bzip2 tools respectively. Both of the commands decompress and feed the file to grep to search. However zgrep and bzgrep don't store the decompressed file. This approach is slightly faster than the approach discussed in Table 4.2. Table 4.8 shows that on average, our approach is about 11.89× faster than zgrep, and about 41.66× faster than bzgrep.

In Table 4.9, we compare our approach for search-and-replace operations. To perform this operation we first search randomly chosen sets of strings in the file F and replace it with a random string, and record the time needed. We then search-and-replace the same randomly chosen sets of strings in the compressed file F_C and record the time needed to perform this operation. We found that since the size of compressed file F_C is smaller than the original file F, the time to perform the search-and-replace is faster on the compressed file. Table 4.9 shows that performing search-and-replace using our approach is about 3.41× faster than performing search-and-replace over plain text file F.

File Name	Absolute (KB)	Relative to our approach	
	Our Approach	gzip	bzip2
f1.txt	149410	0.038	0.097
f2.txt	149400	0.037	0.097
f3.txt	149470	0.037	0.097
f4.txt	149480	0.038	0.097
f5.txt	149470	0.037	0.103
f6.txt	149480	0.038	0.097
f7.txt	149460	0.037	0.096
f8.txt	149470	0.038	0.097
f9.txt	149440	0.037	0.097
f10.txt	149440	0.037	0.097
Average	149452	0.037	0.097

Table 4.5: Peak Memory Used for de-compression, search and the re-compression

In Table 4.10, we report the time to search-and-replace text in a compressed file. In case of gzip and bzip2, we first decompress the compressed file and then perform search-and-replace operation on the plain text followed by a re-compression. Table 4.10 shows that our approach is about 18.32× faster than gzip and 34.70× faster than bzip2 in performing search-and-replace on compressed data F_C.

Finally in (Table 4.11), we report the entropy of plain text files and files compressed using gzip, bzip2 and our approach. We use a python based script [5] to calculate entropy. Entropy measures the randomness of data in a file. In a compressed file, we replace a pattern with fewer bits. Hence, if a data file has higher entropy, it will less likely be compressed further. We report an average entropy of 6.59 for files compressed using our approach as compared to an average entropy of 7.99 in case of gzip and bzip2. The plain text files we use to test our approach have an average entropy of 4.6.

File Name	Absolute (s)	Relative to our approach	
	Our Approach	gzip	bzip2
f1.txt	11.87	0.31	0.56
f2.txt	24.77	0.30	0.55
f3.txt	37.32	0.24	0.46
f4.txt	37.09	0.30	0.55
f5.txt	48.16	0.31	0.55
f6.txt	83.13	0.31	0.56
f7.txt	96.04	0.31	0.56
f8.txt	124.22	0.30	0.54
f9.txt	151.26	0.31	0.56
f10.txt	185.95	0.30	0.55
Average	79.98	0.30	0.54

Table 4.6: Compression Time

The size of our binary is 130,179 bytes (128KB), while the size of the wordlist D is 8,818,863 bytes (8.5MB). The size of the hash map (and the reverse hash map) is approx 12,818,863 bytes (12.22 MB).

File Name	Absolute (s)	Relative to our approach	
	Our Approach	gzip	bzip2
f1.txt	7.25	0.098	0.338
f2.txt	15.50	0.087	0.317
f3.txt	25.15	0.070	0.262
f4.txt	23.54	0.090	0.318
f5.txt	30.41	0.092	0.337
f6.txt	50.95	0.098	0.342
f7.txt	59.38	0.096	0.334
f8.txt	76.18	0.093	0.325
f9.txt	93.36	0.091	0.341
f10.txt	114.05	0.093	0.339
Average	49.58	0.091	0.325

Table 4.7: Decompression Time

File Name	Absolute (s)	Relative to our approach	
	Our Approach	zgrep	bzgrep
f1.txt	0.10	6.56	22.52
f2.txt	0.20	7.04	24.35
f3.txt	0.16	11.10	39.22
f4.txt	0.27	7.51	25.90
f5.txt	0.21	13.86	47.99
f6.txt	0.31	15.20	53.02
f7.txt	0.39	13.76	49.36
f8.txt	0.494	13.77	48.19
f9.txt	0.56	15.70	55.10
f10.txt	0.72	14.50	50.98
Average	0.34	11.89	41.66

Table 4.8: Search Time Compared to zgrep and bzgrep

File Name	Absolute (s)	Relative to our approach
	Our Approach	SR on Plain Text
f1.txt	0.28	3.21
f2.txt	0.53	3.40
f3.txt	1.01	2.51
f4.txt	0.87	3.03
f5.txt	1.17	3.12
f6.txt	1.85	3.71
f7.txt	2.34	3.72
f8.txt	2.91	4.07
f9.txt	3.85	3.83
f10.txt	4.66	3.55
Average	1.95	3.41

Table 4.9: Search-and-Replace Time

File Name	Absolute (s)	Relative to our approach	
	Our Approach	gzip	bzip2
f1.txt	0.28	18.90	36.14
f2.txt	0.53	20.09	38.31
f3.txt	1.01	13.17	26.23
f4.txt	0.87	18.20	34.96
f5.txt	1.17	18.38	34.59
f6.txt	1.85	20.38	38.16
f7.txt	2.34	18.74	35.05
f8.txt	2.91	19.09	35.64
f9.txt	3.85	18.25	34.12
f10.txt	4.66	17.97	33.80
Average	1.95	18.32	34.70

Table 4.10: Search-and-Replace on Compressed File

File Name	Plain Text	gzip	bzip2	Our Approach
f1.txt	4.56	7.99	7.99	6.63
f2.txt	4.59	7.99	7.99	6.63
f3.txt	4.66	7.99	7.99	6.25
f4.txt	4.59	7.99	7.99	6.61
f5.txt	4.60	7.99	7.99	6.63
f6.txt	4.56	7.99	7.99	6.65
f7.txt	4.58	7.99	7.99	6.64
f8.txt	4.64	7.99	7.99	6.59
f9.txt	4.58	7.99	7.99	6.64
f10.txt	4.58	7.99	7.99	6.63
Average	4.60	7.99	7.99	6.59

Table 4.11: Entropy

5. CONCLUSIONS

Data compression is an important problem. With the increasing popularity of remote and cloud-based computation, and the increasing dataset sizes obtained in data science experiments, compression is becoming even more important. Improved compression techniques would not only reduce the size of a data object and reduce data transfer times, but also would reduce the amount of data transferred. Traditional compression techniques achieve healthy compression ratios, but require decompression before a lookup can be performed. This increases the lookup time. In this thesis, we propose a compression technique for plain-text data objects, that uses variable length encoding to compress data. We sort the word-list of possible words in the language based on the statistical frequency of the use of words. Then we encode the words in a data object using the variable length code-words. Words that are not in the dictionary are handled as well. The driving motivation of our technique is to perform significantly faster lookups without the need to decompress the compressed data object. Our approach also facilitates string operations like concatenation, insertion, deletion and search-and-replace on the compressed file itself without the need for decompression. This is important, particularly with increasing data object sizes that are being encountered. We have implement our technique in C++, and compared our approach with industry standard tools like gzip and bzip2 in terms of compression ratios, speed of lookup, search-and-replace time, and peak memory use.

6. FUTURE WORK

One of the applications of our approach would be in industry where huge volume of log files are generated daily. Due to huge volume of log-files, it should be stored in compressed form using our approach, it would be possible to search errors in the compressed form itself. In future work, experiments can be performed to execute our approach on different sizes of log files and benchmark its behavior with increasing file size.

The other area where this work can be extended is homomorphic computing where string operations can be performed over compressed data in an encrypted manner. Homomorphic computing is an area of computing where encrypted data sets are sent to a cloud server that allows computations to be carried out on encrypted data sets. The idea can be extended to first randomly shuffling a dictionary word list within a code-word (i.e reshuffle the ranks of the words that fall under the same code-words) and compressing the data on the cloud. We would use the same compression technique to encode search strings and send it to cloud where search would be performed on compressed data. The cloud will send us the result confirming if it the word exists in the compressed data.

Other future work will involve exploring methods to improve memory utilization used by our approach. New ideas could be explored to better utilization of T_0 code to improve compression ratio. Currently we are using only 3 code-words in T_0 category out of 2^6 possible available code-words.

REFERENCES

[1] Amir, A. and Benson, C. Efficient two-dimensional compressed matching. In *Data Compression Conference, 1992. DCC'92.*, pages 279–288. IEEE, 1992.

[2] Azad, M., Kalam, A., Sharmeen, R., Ahmad, S., and Kamruzzaman, S. An efficient technique for text compression. *Information Management and Business IMB 2005, The 1st International Conference on*, 2005.

[3] Bhadade, U.S. and Trivedi, A. Lossless text compression using dictionaries. *Computer Applications, International Journal of*, 13(8), 2011.

[4] Burrows, M. and Wheeler, D. A block-sorting lossless data compression algorithm. In *Digial Src Research Report*. Citeseer, 1994.

[5] code.activestate.com. Entropy Calculation. http://www.kennethghartman.com/calculate-file-entropy/, 2013.

[6] Gutenberg. Free ebooks by Project Gutenberg. http://www.gutenberg.org/wiki/Gutenberg:The_CD_and_DVD_Project, 2016.

[7] Ho, M.H. and Yen, H.C. A dictionary-based compressed pattern matching algorithm. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pages 873–878. IEEE, 2002.

[8] Huffman, D.A. et al. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[9] Khurana, U. and Koul, A. Text compression and superfast searching. *arXiv preprint cs/0505056*, 2005.

[10] Klein, S.T. and Shapira, D. Pattern matching in Huffman encoded texts. *Information processing & management*, 41(4):829–841, 2005.

[11] Levert, C. bzgrep. http://linux.die.net/man/1/bzgrep, 2016.

[12] Levert, C. zgrep. http://www.sbras.ru/cgi-bin/www/unix_help/unix-man?zgrep+1, 2016.

[13] Nelson, M. Data compression with the Burrows-Wheeler transform. *Dr. Dobb's Journal*, 9:46–50, 1996.

[14] Nishad, P. and Sankar, S. Efficient random sampling statistical method to improve big data compression ratio and pattern matching techniques for compressed data. *International Journal of Computer Science and Information Security*, 14(6):179, 2016.

[15] Ntlworld. Frequency of Punctuation. http://homepage.ntlworld.com/vivian.c/Punctuation/PunctFigs.htm, 2013.

[16] Rissanen, J. Arithmetic codings as number representations. *Acta Polytechnica Scandinavica*, 31:44–51, 1979.

[17] Rissanen, J. and Langdon, G.G. Arithmetic coding. *IBM Journal of research and development*, 23(2):149–162, 1979.

[18] Robinson, A. and Cherry, C. Results of a prototype television bandwidth compression scheme. *Proceedings of the IEEE*, 55(3):356–364, 1967.

[19] Searchbin. Searchbin. https://github.com/Sepero/SearchBin, 2016.

[20] Welch, T.A. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[21] Winwaed.com. Frequency of English words. http://www.winwaed.com/blog/2012/04/16/calculating-word-and-n-gram-statistics-from-a-wikipedia-corpora/, 2012.

[22] Ziv, J. and Lempel, A. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[23] Ziv, J. and Lempel, A. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.